MCRENGINE: A Scalable Checkpointing System Using Data-Aware Aggregation and Compression

Tanzima Zerin Islam*, Kathryn Mohror†, Saurabh Bagchi*, Adam Moody†, Bronis R. de Supinski† and Rudolf Eigenmann*
*School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN
Email: {tislam,sbagchi,eigenman}@purdue.edu
†Lawrence Livermore National Laboratory (LLNL), Livermore, CA
Email: {kathryn,moody20,bronis}@llnl.gov

***Abstract*—High performance computing (HPC) systems use checkpoint-restart to tolerate failures. Typically, applications store their states in checkpoints on a parallel file system (PFS). As applications scale up, checkpoint-restart incurs high overheads due to contention for PFS resources. The high overheads force large-scale applications to reduce checkpoint frequency, which means more compute time is lost in the event of failure.**

We alleviate this problem through a scalable checkpoint-restart system, MCRENGINE. MCRENGINE aggregates checkpoints from multiple application processes with knowledge of the data semantics available through widely-used I/O libraries, e.g., HDF5 and netCDF, and compresses them. Our novel scheme improves compressibility of checkpoints up to 115% over simple concatenation and compression. Our evaluation with large-scale application checkpoints show that MCRENGINE reduces checkpointing overhead by up to 87% and restart overhead by up to 62% over a baseline with no aggregation or compression.

I. INTRODUCTION

As software and hardware component counts in high performance computing (HPC) systems scale up, the likelihood grows of one failing while an application executes. For example, the 100,000 node BlueGene/L system at Lawrence Livermore National Laboratory (LLNL) experiences an L1 cache parity error every 8 hours [1] and a hard failure every 7-10 days. Exascale systems are projected to fail every 3-26 minutes [2], [3]. Many applications tolerate failures through checkpoint-restart [4], [5], which periodically saves application state in *checkpoint* files on stable storage, such as a parallel file system (PFS). If a failure occurs, the application is restarted from the latest checkpoint, thus reducing repeated computation. To simplify checkpoint-restart implementations, many applications have all processes take checkpoints simultaneously [6]. This strategy avoids the complexities of message logging that uncoordinated checkpointing requires, as well as its possibility of cascading rollback.

Checkpointing causes 75-80% of the I/O traffic on current HPC systems [7], [8]. On future systems, checkpointing activities will dominate compute time and overwhelm file system resources [5], [9]. Checkpointing to a PFS is expensive at large scale: a checkpoint can take tens of minutes due to network bandwidth and PFS resource contention [10], [11]. Further, HPC computational capabilities are increasing more quickly than their I/O bandwidths. For example, BlueGene/L at LLNL and BlueGene/P at Argonne National Laboratory achieve less than 0.1 GB/s of I/O bandwidth per TeraFLOP of computational capability [10]–[12]. This high overhead leads to reduced checkpointing frequency, which implies more computation is lost in the event of failure.

We face two key challenges to checkpointing scalability: the number of checkpoint files written and their size. As application process counts grow, the number of checkpoint files usually increases proportionally. Large counts of file writers degrades PFS performance and reliability due to contention [12]–[14]. Thus, application programmers are exploring techniques to combine checkpoints of multiple processes at the cost of application complexity. Further, the size of each file grows with process count under naive combining schemes; in any event, the size of the checkpoint files usually grows as processes compute larger sub-problems.

We address these challenges through MCRENGINE, a library that aggregates multiple checkpoint files from different processes and compresses them. Further, we extract application-specific data semantics by analyzing the metadata contained in the checkpoint files. This knowledge enables us to merge the checkpoint fragments from the different processes intelligently, increasing the compressibility of the aggregated checkpoint, particularly when we apply data-specific compression algorithms (e.g., targeted to floating point data). These benefits also apply during restart, when all application processes must read the checkpoint fragments from the PFS, often within a short time window. We must keep this extra work small since restart is always on the application's critical path. We make the following major contributions:

- The concept of data-aware checkpoint compression, which significantly increases the compressibility of checkpoints;
- A thorough investigation of the application-specific impact of different aggregation schemes on compression ratio;
- The design, development, and thorough evaluation of an end-to-end checkpoint-restart system and its demonstration with three real-world, complex, and diverse applications.

Our evaluation uses ALE3D, Cactus, and Enzo. ALE3D is an arbitrary Lagrangian-Eulerian (ALE) multi-physics code. Cactus [15], [16] solves Einstein's Equations. Enzo is a grid-based code that simulates cosmological structure formation [17]. Our significant results include: **1)** data-aware compression improves compression ratios between 27.72% and 115% on average over simple concatenation and compression; **2)** MCRENGINE can reduce checkpointing intervals by up

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

to 87%; **3)** MCRENGINE decreases restart overhead by 62% compared to using uncompressed checkpoints; **4)** data-aware compression alone reduces the latency to write checkpoints to the PFS up to 92% and to read checkpoints up to 71%.

The paper is organized as follows. Section II provides background. Section III describes our solution and rationale. Section IV details the structure of the implementation of MCRENGINE. Section V presents our evaluation methodology. Sections VI and VII present key results.

II. BACKGROUND

In this section, we discuss related background information on different checkpoint types, how checkpoints are written and the different file formats in which they are written.

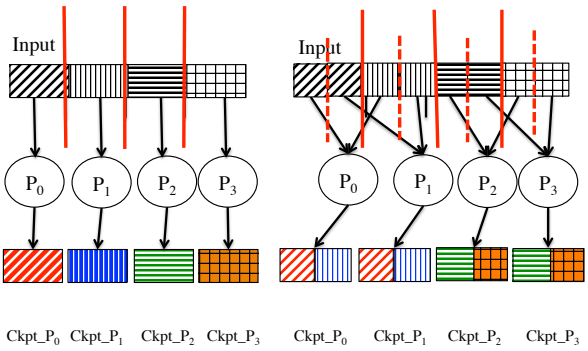
A. Application-level vs System-level Checkpointing

HPC applications commonly take coordinated, application-level checkpoints. Coordination implies that all application processes agree when they will take the checkpoint, typically periodically. Application-level checkpointing uses application routines to save specific data structures in the checkpoint files. In contrast, system-level checkpoints are system-initiated snapshots that include the entire system memory. Application-level checkpointing provides several benefits. Application knowledge of the state needed to restart can significantly reduce checkpoint sizes. For example, for protein-folding applications on IBM Blue Gene, the size of an application-level checkpoint set is a few megabytes compared to terabytes for full system-level checkpoints [18]. Application-level checkpoints also increase portability.

B. Checkpoint Writing

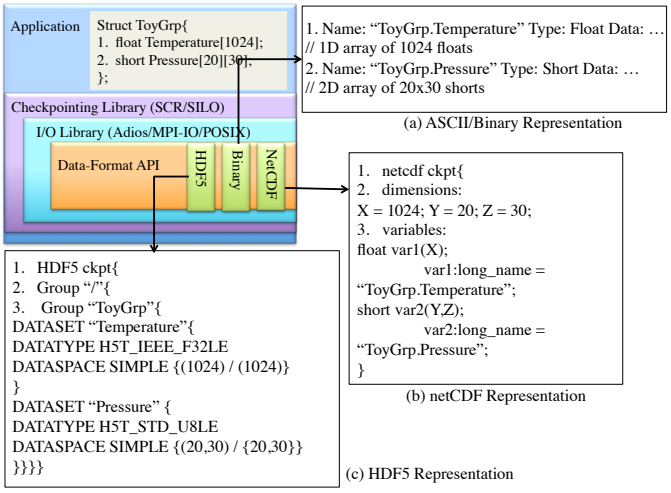
Applications vary in how many processes write checkpoints. If an MPI application uses N processes, then each process writes its own state to a checkpoint file in the $N \rightarrow N$ pattern. This strategy performs poorly at large scales due to contention from the large number of writers. At the other extreme ($N \rightarrow$ 1 checkpointing), a single process writes one large checkpoint that contains data from all processes, which creates a serialization bottleneck that also inhibits scaling. Thus, $N \rightarrow M$ checkpointing, in which a set of M processes write aggregated data for N processes ($M < N$), attempts to balance metadata costs and contention with the bandwidth advantages of multiple writers. However, $N \rightarrow M$ checkpointing increases application complexity, a trend that further optimizations continue. Thus, application developers need a strategy that provides the benefits of $N \rightarrow M$ checkpointing while hiding that complexity. MCRENGINE is such a strategy.

Applications also vary in how they distribute input data across processes. One approach is *segmented input distribution* (Figure 1a), in which data is naturally composed of segments that exhibit similarity. One segment of the input data is processed by one process. The second approach is *strided input distribution* (Figure 1b), in which one segment of the data is split into stripes and these stripes are distributed (or "strided") to different application processes.



(a) Segmented Input Distribution (b) Strided Input Distribution

Fig. 1: Two Types of Input Data Distribution



(a) ASCII/Binary Representation

(b) netCDF Representation

(c) HDF5 Representation

Fig. 2: Different Metadata Annotations of Checkpoint Data

C. Checkpoint File Format

Application-level checkpointing often builds on I/O libraries such as MPI-IO, POSIX, HDF5, netCDF, or Parallel-netCDF [19], [20]. While some applications write checkpoints in an application-specific format, using standard I/O libraries, such as POSIX or MPI-IO, many large-scale applications use standard self-describing I/O formats (e.g., HDF5 or netCDF). While the initial time to deploy POSIX or MPI-IO solutions can be low, it reduces portability due to issues such as endianness and thus increases maintenance cost. Using a descriptive data format to write structured checkpoints ensures portability across users, tools (such as visualizers), and systems. Thus, self-describing I/O formats are popular among large-scale applications, such as HDF5 [19] with its long list of users.

As Figure 2 shows, data structures in application-level checkpoints are described differently depending on the data format. Also checkpoint files are often used for visualization. Although the file format structure varies, the key point is that application data is annotated with descriptive metadata. The application (Figure 2a) or the data format library (Figure 2b and c) can provide the metadata.

MCRENGINE uses data from application-level, globally coordinated checkpoints to implement an $N \rightarrow M$ scheme. To demonstrate our approach, we use HDF5 checkpoints.

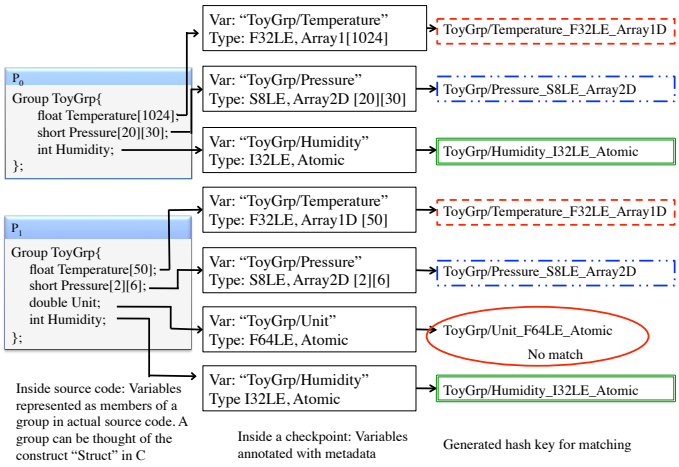


Fig. 3: Variable Matching

However, the design of MCRENGINE provides the flexibility to use other data formats easily. Using the example in Figure 2, data with the name *Temperature* has the same meaning across all processes in an Single Program Multiple Data application (we may require additional information for a Multiple Program Multiple Data application to capture data relationships across processes). We refer to an individual file that a single process writes as a *checkpoint* and to a group of checkpoints that multiple processes write concurrently as a *checkpoint set*.

III. Data-Aware Checkpoint Aggregation & Compression

Most parallel scientific applications distribute simulation data across multiple processes. These processes generally coordinate globally to take a consistent checkpoint. During an asynchronous checkpointing phase, once a process finishes checkpointing, it resumes its computation. Currently, with most checkpointing systems, the processes send their checkpoints directly to the PFS, although checkpointing libraries such as SCR [9] can store checkpoints locally instead.

In order to analyze the issues that will prevent scaling of current checkpointing systems, we conducted an experiment on a large Linux cluster. We present a complete discussion of the experiment in Section VII-B. Our key findings are:

- The average performance of operations (read and write) scales much better with fewer concurrent processes;
- Transfer overhead is lower with the same number of writers when the data volume is less.

Thus, checkpointing systems should aggregate checkpoints to reduce the number of concurrent processes. Compressing checkpoints could provide additional benefits. We define *data-aware aggregation* as aggregating across process checkpoints such that data with similar meaning remain together in the merged checkpoint. MCRENGINE uses *data-aware compression*, which dynamically selects from a set of compression algorithms and applies the best one for each similar data group. *Data-aware aggregation and compression* is the entire process of interleaving semantically similar data and using a dynamically selected compression algorithm.

All popular compression utilities use a finite window in which they look to find similarities between two data items. The window is kept reasonably small by default (e.g., 32KB for gzip) to keep the memory utilization of the compression utility low. By using data-aware aggregation, MCRENGINE increases the likelihood that similar data appears in the same window, which increases compression and decreases the number of concurrent files written to the PFS.

A. Identifying Similarity Across Checkpoints

Intuitively, finding similarity across checkpoints is simple. Two variables with the same name and data type are likely to have the same meaning. MCRENGINE uses metadata to locate checkpoint data that represent the same variables. MCRENGINE interprets two variables in different checkpoints as *similar* if their names agree and their data representation is identical (same data type).

Figure 3 provides a step-by-step example. Processes P_0 and P_1 have groups of variables. To be general, the example shows different structures for the same group in the two checkpoints although in practice, two processes of the same application are unlikely to have different structures for the same group.

In our similarity detection, variables can be of any standard data type and can be of atomic or array class. We consider variables with the same name, data type and class to be similar even if they have different numbers of elements. Variable names can match either exactly or according to a regular expression; application developers can specify the regular expression as a configuration parameter. We locate similarity as follows:

- Match every element of a group separately;
- Annotate variables with their names, data types, array sizes, and classes in the checkpoints;
- Convert variable metadata into hash keys of type: `Group-name/Variable-name_Datatype_Class`.

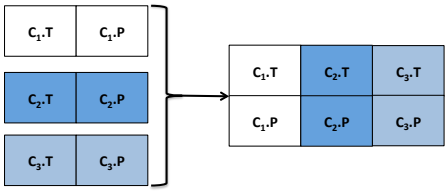
For example, MCRENGINE generates `ToyGrp/Temperature_F32LE_Array1D` as the hash key for the variable `Temperature` in P_0. While Figure 3 illustrates our terminology, real application checkpoints have more complex structures.

B. Merging Schemes

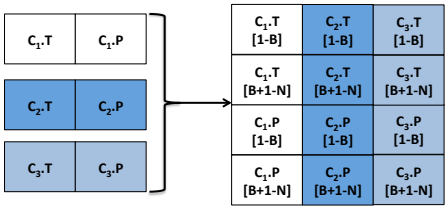
We consider four schemes to aggregate checkpoints:

- **Checkpoint Concatenation** The simple *Agnostic* scheme concatenates entire checkpoints before compressing them;
- **Checkpoint Blocking** *Agnostic-Block* interleaves fixed-size data blocks instead of concatenating entire checkpoints;
- **Variable Concatenation** As Figure 4a shows (Cx.T denotes the temperature array of the checkpoint of rank x and Cx.P its pressure array), the *Data-Aware* or simply *Aware* scheme concatenates individual variables before compressing them;
- **Variable Blocking** As Figure 4b shows, the *Aware-Block* scheme interleaves variables in blocks of configurable sizes, instead of simply concatenating them.

The Aware and Aware-block schemes aim to increase compression ratios by grouping similar data.



(a) Variable Concatenation



(b) Variable Blocking

Fig. 4: Data-Aware Merging Schemes

IV. Structure of mcrEngine

Figures 5a and 5b present the system level interactions among different modules of mcrEngine during the checkpoint and restart phases. In these figures, each large dotted box represents a component and each small box represents a module. A solid module indicates that it is on the critical path; a dotted one indicates it avoids the critical path by using a parallel thread. Finally, arrows represent different inter-module communication: a dashed arrow indicates network communication while a dotted one corresponds to a wake up signal to a thread. mcrEngine has two major components:

- *Compute node component (CNC)* offloads checkpoint processing from the compute nodes by sending checkpoints to the aggregator node component during checkpointing; restores checkpoints to the local disk during restart.
- *Aggregator node component (ANC)* processes (e.g., concatenates and compresses) checkpoints; stores and retrieves checkpoints from the PFS during restart.

We now discuss interactions between CNC and ANC modules during the checkpointing and restart phases.

A. Checkpointing Phase

Table I summarizes the interactions between the CNC and ANC components. CNCs responsible for application processes in the same group (according to their rank) send data to the same ANC. Each CNC notifies the assigned ANC that a checkpoint is ready and transfers the data to it. The ANC concatenates and merges checkpoints based on the scheme in use and writes to the PFS. The message queue is a memory buffer in which each element points to a data block in memory. These lightweight modules require little CPU and memory resources. The CNC uses the Inotify blocking wait to reduce interference with the application.

We design mcrEngine such that its aggregators fetch variables as needed in order to avoid requiring excessive aggregator disk space. By pulling data as needed, the ANC can limit the variables per process present at any time. We also exploit the inherent parallelism of accessing three kinds of resources (CPU, disk, and network) by using separate threads to transfer data between the CNC and ANC modules, to compress it and to write it to disk. Thus, when the ANC `Fetch & Merge` module starts a network transfer (fetch operation), it schedules the `Compressor` thread to use the CPU and the `Post-Processor` thread to finish disk I/O. Our results show that the network transfer time largely offsets the overheads of compression and local disk accesses.

The CNC also produces a metadata file that stores information about the last checkpoint set. The restart phase uses the information about the last set of checkpoints stored. An empty file indicates a new application run.

TABLE I: Module Interactions During a Checkpointing Phase

ID	Communication Description
1	Inotify Library notifies the Controller of checkpoint creation
2	Controller initiates checkpoint reading through CNC interfaces
3	CNC Ckpt Library reads the checkpoints, written in application-specific formats, into memory (3a); returns header and data to the Controller (3b)
4	Controller passes reference to header/data buffer to Network Transceiver
5	Network Transceiver sends information about variables to Header Receiver
6	Header Receiver sends group name, variable name, data type, and class information to Similarity Classifier, which inserts them into a hash table, chaining variables with the same hash key in process rank order (6a); Header Receiver invokes Fetch & Merge Handler after all checkpoint headers of a group are classified (6b)
7	Fetch & Merge Handler signals Compressor to compress variables, if available (7a); traverses hash table to send data requests to Network Transceiver and fetches data for a pool of variables (7b); merges similar variables and enqueues them in a message queue (7c)
8	Compressor signals the Post-Processor thread to start buffering processed data, if available (8a); reads from the message queue; compresses merged variables according to data types (current implementation uses FPC [21], fpzip [22] and LZ [23] to compress doubles, floats, and other data types, respectively) and writes back to the message queue (8b)
9	Post-Processor flushes accumulated data to local disk (9a); applies Parallel-Gzip [24] after all variables across processes in the same group are merged and compressed; sends the final data to PFS (9b)

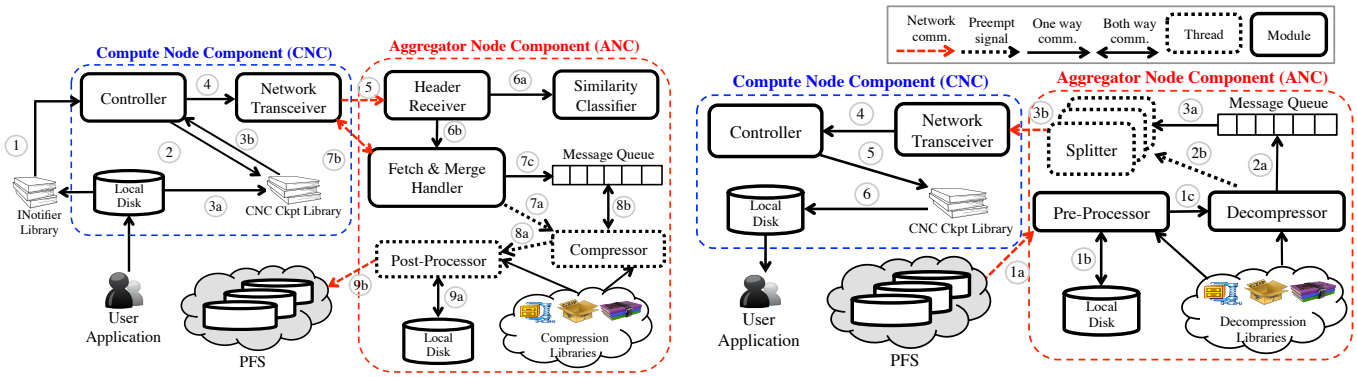
TABLE II: Module Interactions During a Restart Phase

ID	Communication Description
1	Pre-Processor reads processed checkpoint and metadata file from PFS (1a); applies Parallel-unzip and writes to the local disk (1b); invokes Decompressor on merged-compressed variables
2	Decompressor applies appropriate decompression algorithm to each variable; enqueues decompressed merged variables in the message queue (2a); signals Splitter threads to split variables, if available (2b)
3	Splitter reads and separates variables from message queue them (3a); uses a separate thread per process to send data to the Network Transceivers (3b)
4	Network Transceiver invokes the Controller module with received data
5	Controller invokes interfaces to write variables to checkpoint file
6	CNC writes variables to checkpoint file in application-specific format

B. Restart Phase

As Figure 5b shows, during restart, the `Controller` module sends a request to the `Pre-Processor` module of the corresponding ANC to read a checkpoint file from the PFS. The `Pre-Processor` module then reads the checkpoint from the PFS, decompresses and separates the variables, and sends them to the `Controller`. Each `Splitter` thread communicates in parallel with a separate `Controller`, which must generate the original checkpoints for the application processes. Table II summarizes the communications between CNC and ANC modules during this phase.

The design of mcrEngine does not require node-local persistent storage on compute or aggregator nodes. This design



(a) MCRENGINE Checkpoint Phase Interactions

(b) MCRENGINE Restart Phase Interactions

Fig. 5: System-Level Interactions

feature allows MCRENGINE to work on current large-scale systems (e.g., BG/L). On diskless nodes, the CNC uses in-memory buffering before streaming them to the ANC next stop without any intermediate storage (*checkpoint streaming*). However, upcoming technologies such as SSDs will replace the in-memory buffer. Our future work will implement *variable streaming*, in which we buffer and stream large variables individually. That work will address issues such as synchronizing similar variable writes.

V. Evaluation Methodology

This section describes our methodology for evaluating MCRENGINE with the schemes introduced in Section III-B. We evaluate MCRENGINE with checkpoints from four computational simulations of three real-world applications. Our evaluation criteria are the amount of compression, and checkpoint-restart end-to-end overhead.

A. Applications

We evaluate MCRENGINE on ALE3D, Cactus, and Enzo (with two distinct inputs: *Cosmology* and *Implosion*). The simulations produce checkpoints with varying characteristics. Table III summarizes our ALE3D checkpoint set and a checkpoint set from the midpoint of the Cactus and Enzo simulations. The write pattern indicates how the checkpoints are written. The number of checkpoints indicates how many individual checkpoints are in each set. For ALE3D, 1024 computation processes were run to generate 32 checkpoints. For the other applications, the number of checkpoints generated equals the number of computation processes. The total size is the sum of the individual checkpoint sizes. The individual size row shows the range and average size of the individual checkpoints. We report the total number of variables across a checkpoint set and the range and average of variable counts in the individual checkpoints. We also provide percentage breakdowns of variable data types.

ALE3D is a multiphysics numerical simulation that uses arbitrary Lagrangian-Eulerian (ALE) techniques. We do not have access to the ALE3D source code so we do not have information on its input distribution.

Cactus [15], [16] is a framework that numerically solves Einstein's equations [25]. It uses adaptive mesh refinement (AMR). During initialization, it divides input data among N processes (*top-level roots*). Each process then recursively divides its pieces among sub-grids. Each top-level root collects computation from sub-grid elements and writes a checkpoint. So, this application writes N checkpoints, while the number of sub-processes that perform actual computation is larger.

Enzo, an AMR, grid-based hybrid code (hydro + N-Body), simulates cosmological structure formation [17]. It uses the particle-mesh technique to solve dark matter N-body dynamics. Cosmology is a 3D Eulerian block-structured AMR simulation [26]. Implosion is a 2D converging shock problem in which a shock wave interacts with reflecting walls, undergoing a double Mach reflection [27]. While we generate both from the same code base, the nature of the execution and, thus, the checkpoints are significantly different.

B. Evaluation of Compression Schemes

Two metrics measure how well data-aware compression techniques reduce data size. *Compression ratio* is the ratio of the size of the (checkpoint's) uncompressed data to the size of the compressed data, including metadata. A compression ratio of 2 indicates that the uncompressed file is 2× larger than the compressed file. In Equation 1, $us(ckpt)$ is the uncompressed checkpoint size. For a scheme X, $cs_x(ckpt)$ is its compressed checkpoint size and $cr(X)$ is its compression ratio.

$$cr(X) = \frac{us(ckpt)}{cs_x(ckpt)} \quad (1)$$

Relative improvement compares the effectiveness of compression schemes as the compression ratio of schemes A and B:

$$\text{relative improvement} = \frac{cr(A) - cr(B)}{cr(B)} \times 100\% \quad (2)$$

Finally, we evaluate the performance of the different schemes, which we measure in terms of the time to complete the checkpointing and restart phases.

TABLE III: Checkpoint Characteristics

	ALE3D	CACTUS	COSMOLOGY	IMPLOSION
Write Pattern	N→M	N→N	N→N	N→N
Number of Checkpoints	32	32	128	64
Total Size (GB)	4.8	2.41	1.1	0.013
Individual Size: [Range] Average (MB)	[154.1, 154.5] 154.2	[58, 90] 77.15	[4.7, 13.4] 8.4	[0.07, 4] 0.2
Total Variables	56820	284800	33562	9996
Variable Count: [Range] Average	[1760, 1835] 1776	[8900, 8900] 8900	[76, 1090] 262	[28, 832] 156
Double Precision Floats (%)	88.8	33.94	24.3	0
Single Precision Floats (%)	2e-5	0	67.2	74.1
Other Data Types (%)	11.2	66.06	8.5	25.9

VI. DATA-AWARE COMPRESSION EFFECTIVENESS

We now evaluate the effectiveness of our novel data-aware compression techniques. In particular, we explore:

- The benefit of multiple compression passes;
- The change in compression ratio with varying group size;
- The impact of interleaving granularity on compression ratio;
- The change in compression ratio as a simulation progresses.

We ran our experiments on LLNL's Sierra system [28], which is a 261.3 TFLOP/s Linux cluster running the CHAOS 4.4 operating system with an InfiniBand QDR interconnect. It has 1,944 nodes, each with 12 2.8 GHz cores and 24 GB of memory. Sierra is connected to a 1.3 PB Lustre file system with a maximum aggregate bandwidth of 30 GB/s.

To determine parameters that provide a high compression ratio with low time overhead we evaluated the compression libraries that we used. For each library, we choose settings for each algorithm that realize most of the possible compression without incurring significant overhead. Throughout the experiments, we set the compression levels of Parallel-Gzip to 6 (the default value), of FPC to 2, of fpzip to 1-dimensional, and of QuickLZ to 1 (the minimum value).

For this evaluation, we group application processes according to their rank order. For example, processes 1 to 32 and 33 to 64 are aggregated for a $GROUP_SIZE$ of 32. We run one application process per core on each compute node, and use all cores on each aggregator node for our experiments. Each aggregator process on a core handles checkpoints from $GROUP_SIZE$ computation processes. We use Equation 3 to determine the aggregator node count:

$$\text{\# of Agg-Node} = \frac{cp}{ac \times GROUP_SIZE} \tag{3}$$

where cp is the number of computation processes and ac is the number of cores on each aggregator node.

A. Benefit of Multiple Passes of Compression

We first study the impact of multiple compression passes. We experiment with one and two compression passes, to which we refer as single (SC) and double (DC) compression. SC applies one pass, either with Gzip or data-aware compression. DC always applies Gzip in the second pass. Figure 6 shows the results for *Agnostic*, *Aware*, and *Aware-Block*. We do not show *Agnostic-Block* in Figure 6 because it performs similarly to *Agnostic* for DC. DC never benefits the data-agnostic schemes. Alternatively, DC improves the compression ratio of the data-aware schemes considerably because the compression

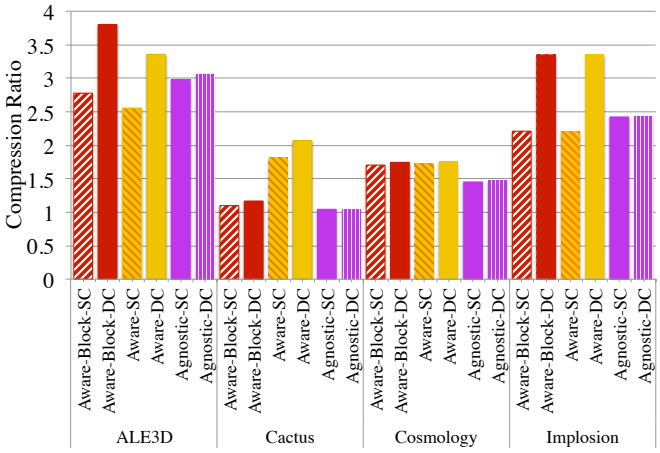


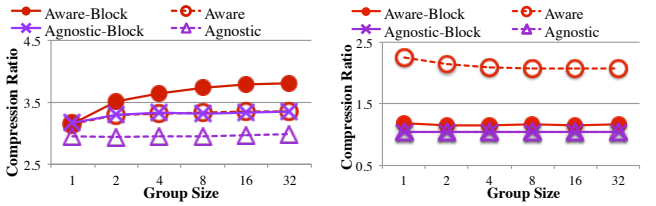
Fig. 6: Double Compression Versus Single Compression

libraries that we use in the first pass convert the data to a more compressible format, which the second pass exploits. For example, FPC applies an XOR operation to a predicted value and the actual value. It encodes the resulting number of zeros, an integer, into the output, which can inflate the size of the output but enables Gzip to compress the transformed data better. In our remaining results, we use double compression for *Aware* and *Aware-Block*; single compression for *Agnostic*.

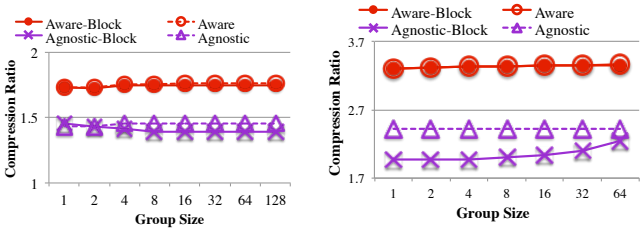
B. Change in Compression Ratio with Varying Group Size

In this section, we evaluate the impact of group size on the compression ratios of the different schemes. We make several observations from Figure 7 (*Aware* and *Aware-Block* overlap in b, c, and d). Overall, we find that *Aware* obtains significant compression for all four simulations. It reduces the total data to about half or even to one-third of the amount compared with the uncompressed checkpoint. Additionally, we see that data-aware compression achieves higher compression ratios than data-agnostic compression. The most dramatic case is for Cactus in Figure 7b for which the gain is 115% by *Aware* compared to *Agnostic*. We also find that changing group size impacts the simulation that benefits most from the *Aware-Block* scheme (ALE3D) more than those that benefit most from the *Aware* schemes (Cactus, Cosmology, Implosion). For ALE3D, the *Aware-Block* scheme achieves an 8% improvement in compression ratio from group size 2 to 32.

We also observe that the best compression scheme varies across applications. For ALE3D, *Aware-Block* is the best, while *Aware* is the best for Cactus. Both data-aware schemes



(a) ALE3D - Block Size 8192B (b) Cactus - Block Size 1024B



(c) Cosmology - Block Size 4096B (d) Implosion - Block Size 4096B

Fig. 7: Compression Ratio Versus Group Size (*Agnostic* and *Agnostic-Block* Overlap in b; *Aware* and *Aware-Block* in c and d)

perform well for Cosmology and Implosion. The segmented data distribution and collection in Cactus, Cosmology and Implosion apparently emulates blocking for these simulations. To determine which scheme to apply to a given application, one could try all schemes offline on a sample checkpoint set and then configure MCRENGINE to use the best option.

Our results for *Aware-Block* and *Agnostic-Block* use the best block size from our tests for each simulation. While *Agnostic-block* could emulate data-aware compression if variable layouts are similar across checkpoints, our results for Cactus, Cosmology and Implosion disprove this theory. Further, while *Agnostic-Block* achieves similar compression ratios to *Aware* for ALE3D, blocking the interleaved variables, as with *Aware-Block*, provides significant additional benefit.

C. Impact of Interleaving Granularity on Compression Ratio

We now discuss the impact of varying block size on the compression ratios that the blocking schemes achieve. In this experiment, we only use checkpoints from ALE3D, since we observed in Figure 7, that only its checkpoints benefit from merging in blocks. Figure 8 shows that the compression ratio increases steadily with *Agnostic-Block*, while with *Aware-Block* it is fairly steady with a small initial increase.

D. Change in Compression Ratio as Simulation Progresses

To evaluate how the compression schemes fare as the simulation progresses through time, we plot the compression ratio for a series of consecutive checkpoint sets in Figure 9, which shows the results for the Cactus, Cosmology, and Implosion simulations. Cactus wrote 21 checkpoint sets, each 5 minutes apart. The Cosmology simulation wrote 12 checkpoint sets each 5 minutes apart. The Implosion simulation wrote 8 checkpoint sets, each 0.05 seconds apart. In all three simulations,

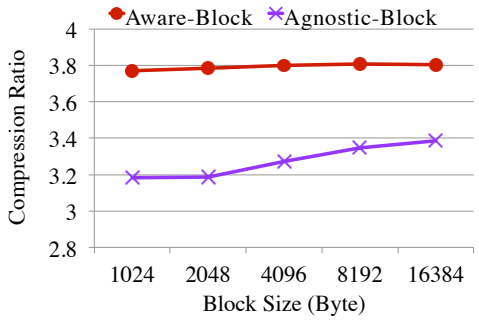


Fig. 8: ALE3D Compression Ratio ($Group_Size$ = 32)

we label the checkpoints as "DD00XX", where "XX" records the simulation time step.

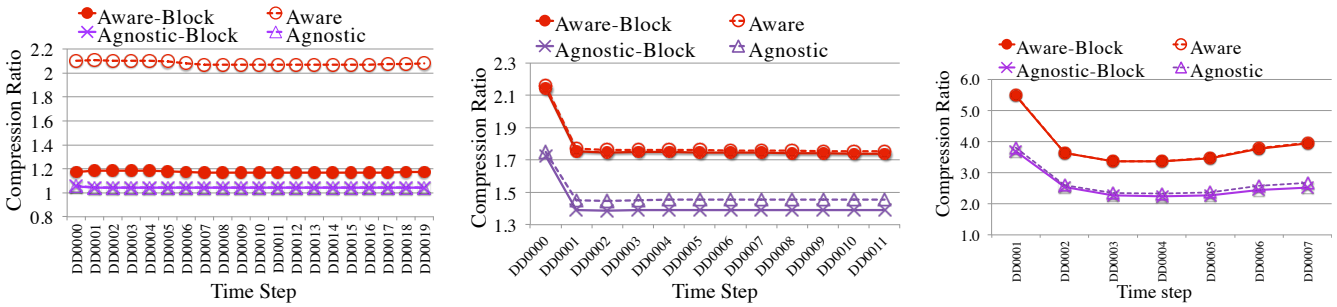
We make three observations from Figure 9. First, the *Aware* and *Aware-Block* schemes yield higher compression ratios than *Agnostic* or *Agnostic-Block*. For example, the relative improvement in compression ratio of Cactus in Figure 9a is about 98%. Second, for both Enzo simulations, the compression ratio of the first checkpoint set is higher than for subsequent ones. The maximum compression ratio for Cosmology drops around 20% (from 2.15 to 1.78) after the first time step. For Implosion, it drops about 50% (from 5.4 to 3.6). These drops occur because Enzo initializes data structures with the same default values so the data is highly compressible across processes while Cactus does not always initialize data structures with default values. Third, the compression ratio quickly levels out as simulations progress. For Cosmology, the compression ratio only has a slight downward slope after the initial drop. Similarly, for Implosion, the compression ratio drops, levels out, and then begins to edge higher as the Implosion simulation runs. We expect the compression ratios to behave in this manner for applications that converge to a result.

E. Summary of Data-Aware Compression Effectiveness

We summarize our evaluation of *Aware* and *Aware-Block* compared to *Agnostic* in Table IV, which shows the largest and smallest relative improvements, which occur with the group size specified in parentheses. While the group size that provides the greatest benefit varies, data-awareness always results in higher compression ratios. We observe that applications with higher interprocess checkpoint similarity gain more compression by interleaving variables in blocks (*Aware-Block*), while applications with higher intraprocess checkpoint similarity gain more by concatenating variables before compressing (*Aware*).

VII. MCRENGINE PERFORMANCE

This section evaluates MCRENGINE overall performance. For checkpointing, MCRENGINE merges and compresses checkpoints and then transfers them to the PFS. For restart, it reads checkpoints from the PFS and then decompresses and splits them into their original format. We demonstrate the efficiency of MCRENGINE for the two largest checkpoint sets, ALE3D and Cactus. For each application, we use a group size



(a) Cactus: Block Size 1024B, Group Size 32 (b) Cosmology: Block Size 4096B, Group Size 64 (c) Implosion: Block Size 1024B, Group Size 64

Fig. 9: Compression Ratio Over Time (*Agnostic* and *Agnostic-Block* Overlap in a and c; *Aware* and *Aware-Block* in b and c)

TABLE IV: Relative Improvement over *Agnostic*

APPLICATION	MEASURE	AWARE-BLOCK	AWARE
ALE3D	Best	**27.72%** (32)	12.7% (32)
	Worst	6.6% (1)	6.6% (1)
Cactus	Best	11.85% (32)	**115%** (1)
	Worst	10.69% (2)	98% (32)
Cosmology	Best	20.59% (1)	**21.14%** (32)
	Worst	20.13% (32)	20.59% (1)
Implosion	Best	38.43% (32)	**38.83%** (32)
	Worst	36.26% (1)	36.26% (1)

TABLE V: Checkpoint/Restart System Configurations

NAME	DESCRIPTION
NoC+NoAgg	No compression, no aggregation
C+NoAgg	Individual compression, no aggregation
NoC+Agg	No compression, aggregation
Agnostic+Agg	MCRENGINE aggregation, data-agnostic compression
Aware+Agg	MCRENGINE aggregation, data-aware compression

of 32, and the data-aware merging scheme from Table IV that provides the best compression ratio (*Aware-Block* for ALE3D and *Aware* for Cactus). Table V describes the different merging and compression schemes that we compare in this section.

The first compression pass yields little benefit for small but compressible variables so we omit them from it. In our framework, the minimum size is configurable; our experiments omit variables $< 100B$. Easily compressed data types include integers and characters. This choice reduces compression ratios by less than 1%, and improves performance.

We measure the times to transfer checkpoints to the parallel file system using IOR [14], [29], a benchmark that characterizes HPC I/O performance. We use the average respective file sizes for each of the compressed or uncompressed checkpoints.

A. Data-aware and Data-Agnostic I/O Performance

We compare the performance of data-aware and data-agnostic compression in terms of time to transfer checkpoints to the PFS. The objective is to study the benefit of having data-aware compression on I/O performance. Data-aware compression results in smaller files and thus requires less time to write the checkpoints to the PFS. The merged and compressed checkpoint sizes per writer for ALE3D were 1.3 GB and 1.5 GB for data-aware and data-agnostic compression respectively. For Cactus, the sizes per writer were 1.2 GB and 2.4 GB.

We show the results for Cactus in Figure 10. In the figure, because the group size is 32, the number of writers

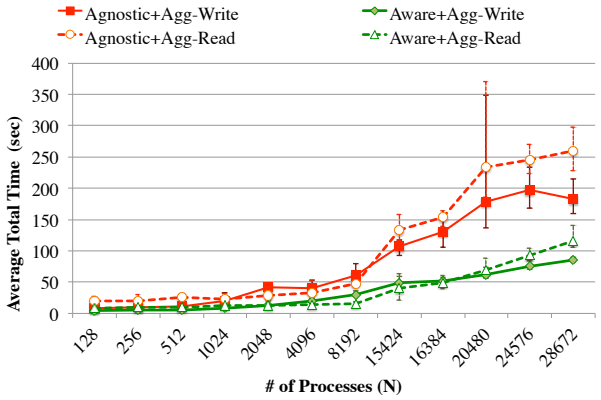


Fig. 10: I/O Performance with Different Compression Schemes

is $N/32$, where N is the number of processes in the job. MCRENGINE with data-aware compression reduces the data transfer overhead compared to data-agnostic compression for both reading and writing checkpoints.

B. Benefit of Checkpoint Aggregation

This experiment shows the PFS I/O performance impact of aggregation as we scale up the application. Figure 11 shows the average measurements and 95% confidence intervals for the time to write to the PFS using IOR for the Cactus checkpoints. For the schemes that aggregate, the number of writers is $N/32$ and the data transferred per reader/writer is 1.2 GB; for the schemes that do not aggregate, the number of writers is N and the data transferred is 87 MB per reader/writer. We find that aggregating the checkpoints in MCRENGINE results in better I/O scalability and reduces performance variability compared to non-aggregated I/O, particularly for read (i.e., restart) operations. Although unclear due to the overhead increase with C+NoAgg-read, Figure 11 shows that aggregation improves write performance up to 26%.

C. Checkpoint and Restart Overheads

We now compare processing and transfer overhead using the MCRENGINE checkpoint and restart schemes. Figures 12 and 13 show the best case average of the measured values of IOR for an application run with 15,408 processes, collected over several days at different times. The numbers beside each step correspond to the steps in Table I and II.

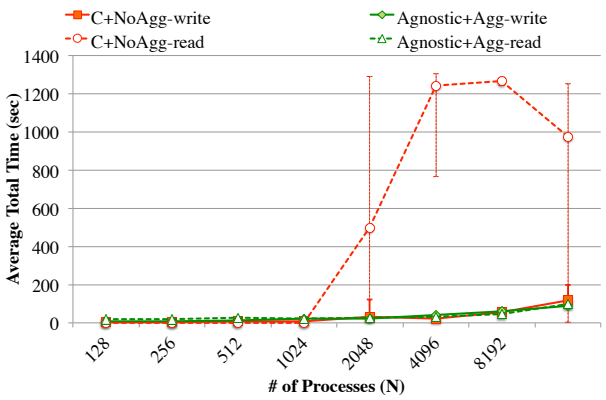


Fig. 11: Benefit of Aggregation for I/O Performance

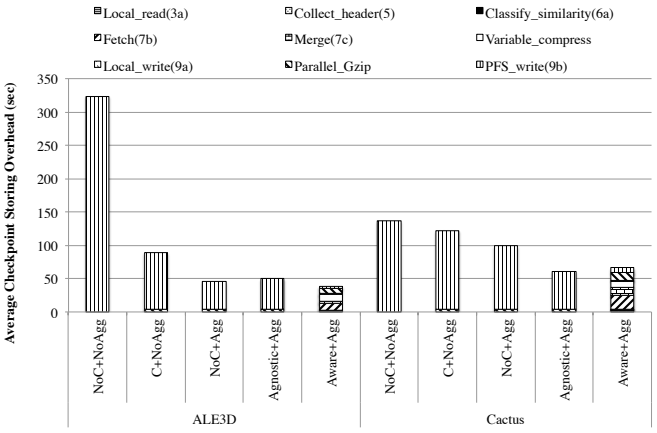


Fig. 12: End-to-End Checkpointing Overhead

We make several observations for checkpointing. First, data-aware compression takes more computation time than data-agnostic and the schemes that only use Gzip or do not compress. However, the checkpoint phase does not occur in the appliation's critical path so that overhead does not affect performance. Second, fetch time for Cactus is higher than ALE3D because it has 5× more variables. Third, the smaller file sizes that result from data-aware compression mean that MCRENGINE uses far less network resources when transferring these checkpoints. Data-aware compression reduces the I/O time by 92% and 86% over data-agnostic (Agnostic+Agg vs Aware+Agg) for ALE3D and Cactus. Finally, MCRENGINE reduces checkpointing overhead by up to 87% over cases without aggregation (NoC+NoAgg vs Aware+Agg for ALE3D), and up to 32% for cases without compression (NoC+Agg vs Aware+Agg for Cactus). This reduction allows applications to checkpoint more frequently, which improves fault-tolerance.

Figure 13 shows the restart overhead results. Since restarts are on the application's critical path, lower overhead implies better end-to-end performance. We again make several observations. Data-aware compression reduces restart overhead by more than 62% compared to the current state of the practice (NoC+NoAgg vs, Aware+Agg) for ALE3D. It reduces overhead by more than 80% compared to only compressing individual checkpoints (C+NoAgg vs Aware+Agg). Second, network transfer time dominates restart overhead for cases without aggregation, compared to CPU time for the MCRENGINE data-aware scheme. Converting a network-bound problem to a CPU-bound problem improves scalability. Finally, data-aware compression significantly reduces network load by reading less data from the PFS compared with the data-agnostic case. The network I/O time for data-aware compression is 43% and 71% less than that of data-agnostic (Agnostic+Agg vs Aware+Agg) compression for ALE3D and Cactus.

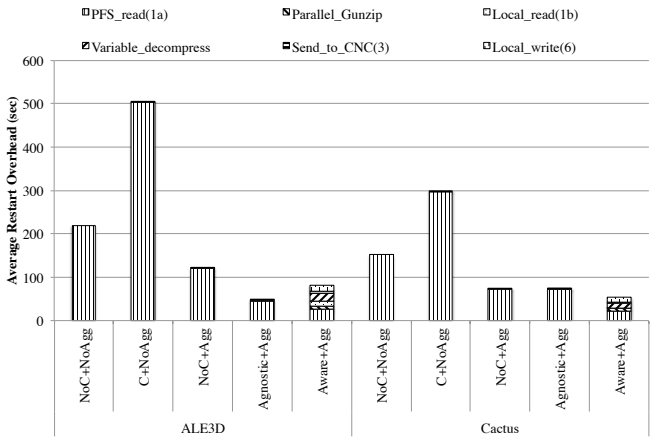


Fig. 13: End-to-End Restart Overhead

D. Discussion

We observe writing to the PFS incurs the largest portion of the total overhead. Thus, our selection of the scheme that provides the highest compression ratio results in the least write time, which makes our scheme selection independent of system characteristics for large checkpoint files. However, system specifications such as I/O, network, and CPU characteristics, will affect the scheme's performance with smaller checkpoint files when computation overhead dominates network overhead. In that case, another selection metric could be the ratio between compression and operation time overhead.

MCRENGINE could be made *adaptive* by evaluating schemes periodically and selecting the best one for that checkpoint set. Also, MCRENGINE currently selects the compression algorithm for each data type based on published results. An adaptive MCRENGINE could sample their compression ratios to select these algorithms dynamically.

We used checkpoints from a variety of runs – both large and small. The ALE3D checkpoint set (the largest set), in particular, is from a production run and shows that the benefits of MCRENGINE increase with checkpoint size. The smaller checkpoints (from Cosmology and Implosion) show less benefit. Currently, we use input sets provided by the community. Configuring these applications to run longer and to take larger checkpoints requires application-specific knowledge.

VIII. Related Work

Other researchers have investigated reducing the size of checkpoints by writing less data in the checkpoints or by com-

pressing the checkpoint files. Incremental checkpointing reduces the size of checkpoints by writing changes in application data between full checkpoints [30]–[33]. These approaches are orthogonal to our work, as incremental checkpoints can be compressed for further savings [34].

Plank and Li compressed checkpoints to fit them in main memory [35], using a variation of Algorithm 1 [36]. They reported compression factors of 10 to 96 percent. They found that compression was beneficial with large and highly compressible checkpoints and when the compression factor exceeded the ratio of the disk write speed to compression speed. Li and Fuchs did not find any advantage in compressing checkpoints with an LZW algorithm on a uniprocessor system despite observing good compression factors since compression time exceeded uncompressed write time [37]. Islam et. al. [38] found that compression and decompression overheads dominated checkpointing and restart times but the decreased network transfer time of the smaller files outweighed the higher overheads, especially with increasing checkpoint size. Ansel et. al. [39] also found that the time to compress checkpoints resulted in longer checkpoint times when writing to node-local disks although the disparity in restart times was smaller. These approaches all use data-agnostic compression while we investigate data-aware techniques. To the best of our knowledge, we are the first to investigate a data-aware compression approach and cross-process merge and compression for parallel applications with many processes.

Bautista-Gomez et. al. [40] developed a topology-aware reliable checkpointing library that provides reliability in cluster environments, which is different from the scalability challenge that we address. Wu et. al. [41] developed a compression scheme for bitmaps and demonstrated that it reduces bitmap index sizes and database query response time.

Several researchers have explored reducing the cost of checkpointing and writing to the file system. To reduce the number of writes to the file system, and ultimately to reduce checkpoint writing cost, Ouyang et. al. [42] cache writes of small and medium sizes within a node in order to aggregate them. Bautista-Gomez et. al. [43] avoid the I/O bottleneck of disk-based checkpointing and the issues of classic diskless checkpointing. These approaches are complementary to ours. They reduce the cost of writing data while our work reduces the amount of data written.

IX. Conclusion

We presented mcrEngine, a novel scheme to compress checkpoints across processes. mcrEngine exploits semantic information in checkpoints to put similar data, i.e., with the same data type and name, close together to improve compression. We implemented several different schemes to interleave data from multiple checkpoints and found that different schemes provide the best results for different applications.

We evaluated mcrEngine and the different interleaving schemes through four real-world computational simulations in terms of the compression ratio and overhead. We found that the amount of compression gained from each scheme varied depending upon how the data was laid out across multiple processes. For some applications, similar data occur in the checkpoint from a single process, while for others, the similar data are striped in blocks and distributed across multiple processes. Thus, the best approach varies. Our compression schemes are as much as 115% better in reducing checkpoint size than simply applying Gzip compression to concatenated checkpoints. As expected, mcrEngine spends more computational time than simply applying compression; however, transferring less data overall with fewer concurrent writers reduced the overhead of writing to the parallel file system. mcrEngine converts network load to computational load; a fact that is crucial for making mcrEngine scale well with increasing application size. Since the parallel file system is often the bottleneck, this method reduces checkpointing overhead by up to 87%. Further, smaller files reduce restart times by up to 62%, a benefit that is particularly important since they occur on the application's critical path. In conclusion, by reducing the time to store and retrieve checkpoints, mcrEngine makes checkpointing-based fault-tolerance in large systems practical.

X. Acknowledgement

LLNL-CONF-554251: This article has been authored by Lawrence Livermore National Security, LLC under Contract No. DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for U.S. Government purposes.

This work was also supported, in part, by the National Science Foundation under grant No. 0833115-CCF.

References

[1] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. Gunnels, and F. Streitz, "Extending Stability Beyond CPU Millennium: a Micron-Scale Atomistic Simulation of Kelvin-Helmholtz Instability," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. ACM, 2007, p. 58.

[2] B. Schroeder and G. A. Gibson, "Understanding Failures in Petascale Computers," in *Journal of Physics: Conference Series*, vol. 78. IOP Publishing, 2007, p. 012022.

[3] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, and K. Hill, "ExaScale Software Study: Software Challenges in Extreme Scale Systems," *DARPA IPTO, Air Force Research Labs, Tech. Rep*, 2009.

[4] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[5] E. N. Elnozahy and J. S. Plank, "Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 97 – 108, April-June 2004.

[6] P. Lemarinier, A. Bouteiller, G. Krawezik, and F. Cappello, "Coordinated Checkpoint Versus Message Log for Fault Tolerant MPI," *International Journal of High Performance Computing and Networking*, vol. 2, no. 2, pp. 146–155, 2004.

[7] F. Petrini, "Scaling to Thousands of Processors with Buffer Coscheduling," in *Scaling to New Height Workshop*, Pittsburgh, PA, 2002.

[8] "The ASC Sequoia Draft Statement of Work," https://asc.llnl.gov/sequoia/rfp/02_SequoiaSOW_V06.doc, 2008.

[9] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10*, November 2010, pp. 1 –11.

[10] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O-Forwarding Infrastructure for Petascale Architectures," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2008, pp. 153–162.

[11] R. Ross, J. Moreira, K. Cupps, and W. Pfeiffer, "Parallel I/O on the IBM Blue Gene/L System," *Blue Gene/L Consortium Quarterly Newsletter, Tech. Rep., First Quarter*, 2006.

[12] R. Hedges, B. Loewe, T. McLarty, and C. Morrone, "Parallel File System Testing for the Lunatic Fringe: The Care and Feeding of Restless I/O Power Users," in *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies*. IEEE Computer Society, 2005, pp. 3–17.

[13] T. H. Cormen and D. Kotz, "Integrating Theory and Practice in Parallel File Systems," in *Proceedings of the 1993 DAGS/PC Symposium*, vol. 7, 1993.

[14] H. Shan and J. Shalf, "Using IOR to Analyze the I/O Performance of XT3," in *Proceedings of the 49th Cray User Group (CUG) Conference*, 2007.

[15] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf, "The Cactus Framework and Toolkit: Design and Applications," in *Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science*. Berlin: Springer, 2003. [Online]. Available: http://edoc.mpg.de/3341

[16] G. Allen, W. Benger, T. Dramlitsch, T. Goodale, H. C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf, "Cactus Tools for Grid Applications," *Cluster Computing*, vol. 4, no. 3, pp. 179–188, 2001.

[17] B. O'Shea, G. Bryan, J. Bordner, M. Norman, T. Abel, R. Harkness, and A. Kritsuk, "Introducing Enzo, an AMR Cosmology Application," *Adaptive Mesh Refinement – Theory and Applications*, pp. 341–349, 2005.

[18] G. Bronevetsky, K. Pingali, and P. Stodghill, "Experimental Evaluation of Application-Level Checkpointing for OpenMP Programs," in *Proceedings of the 20th Annual International Conference on Supercomputing*, 2006, pp. 2–13.

[19] "Who Uses HDF?" http://www.hdfgroup.org/users.html.

[20] "The NetCDF Users Guide," http://www.unidata.ucar.edu/software/netcdf/docs/netcdf.pdf.

[21] M. Burtscher and P. Ratanaworabhan, "FPC: A High-speed Compressor for Double-Precision Floating-Point Data," *IEEE Transactions on Computers*, pp. 18–31, 2008.

[22] P. Lindstrom and M. Isenburg, "Fast and Efficient Compression of Floating-Point Data," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 5, pp. 1245–1250, 2006.

[23] L. Reinhold, "QuickLZ," 2009.

[24] "A Parallel Implementation of GZIP for Modern Multi-processor, Multi-core Machines," http://zlib.net/pigz/.

[25] E. Seidel and W. Suen, "Numerical Relativity as a Tool for Computational Astrophysics," *Journal of Computational and Applied Mathematics*, vol. 109, no. 1-2, pp. 493–525, 1999.

[26] J. Li, W. Liao, A. Choudhary, and V. Taylor, "I/O Analysis and Optimization for an AMR Cosmology Application," in *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*. IEEE, 2002, pp. 119–126.

[27] R. Liska and B. Wendroff, "Comparison of Several Difference Schemes on 1D and 2D Test Problems for the Euler Equations," *SIAM Journal of Scientific Computing*, vol. 25, no. 3, pp. 995–1017, 2003.

[28] "Capability Cluster Sierra at LLNL," https://computing.llnl.gov/?set=resources&page=OCF_resources#sierra.

[29] H. Shan, K. Antypas, and J. Shalf, "Characterizing and Predicting the I/O Performance of HPC Applications Using a Parameterized Synthetic Benchmark," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, p. 42.

[30] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive Incremental Checkpointing for Massively Parallel Systems," in *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*, 2004, pp. 277–286.

[31] S. I. Feldman and C. B. Brown, "IGOR: A System for Program Debugging via Reversible Execution," in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD)*, 1988, pp. 112–123.

[32] N. Naksinehaboon, Y. Liu, C. B. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott, "Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, 2008, pp. 783–788.

[33] K. Ferreira, R. Riesen, R. Brighwell, P. Bridges, and D. Arnold, "libhashckpt: Hash-Based Incremental Checkpointing Using GPUs," *Recent Advances in the Message Passing Interface*, pp. 272–281, 2011.

[34] J. S. Plank, J. Xu, and R. H. B. Netzer, "Compressed Differences: An Algorithm for Fast Incremental Checkpointing," University of Tennessee, Tech. Rep. CS-95-302, August 1995.

[35] J. S. Plank and K. Li, "ickp: A Consistent Checkpointer for Multicomputers," *IEEE Parallel & Distributed Technology*, vol. 2, no. 2, pp. 62–67, 1994.

[36] M. Burrows, C. Jerian, B. Lampson, and T. Mann, "On-line Data Compression in a Log-Structured File System," *ACM SIGPLAN Notices*, vol. 27, pp. 2–9, September 1992.

[37] C. Li and W. Fuchs, "CATCH - Compiler-Assisted Techniques for Checkpointing," in *20th International Symposium on Fault-Tolerant Computing*, June 1990, pp. 74–81.

[38] T. Z. Islam, S. Bagchi, and R. Eigenmann, "FALCON: A System for Reliable Checkpoint Recovery in Shared Grid Environments," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, pp. 1–12.

[39] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop," in *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[40] L. A. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High Performance Fault Tolerance Interface for Hybrid Systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2011, pp. 1–12.

[41] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing Bitmap Indices with Efficient Compression," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 1, pp. 1–38, 2006.

[42] X. Ouyang, K. Gopalakrishnan, and D. K. Panda, "Accelerating Checkpoint Operation by Node-Level Write Aggregation on Multicore Systems," in *International Conference on Parallel Processing*. IEEE, 2009, pp. 34–41.

[43] L. A. Bautista-Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, "Distributed Diskless Checkpoint for Large Scale Systems," in *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 63–72.